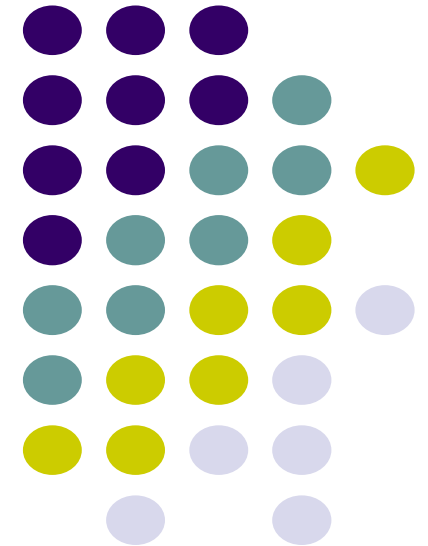


The Basics of UNIX/Linux

13-1. Struct

Instructor: Joonho Kwon
jhkwon@pusan.ac.kr
Data Science Lab @ PNU



Lecture Outline



- `structs` and `typedef`

Structured Data



- A `struct` is a C datatype that contains a set of fields
 - Similar to a Java class, but with no methods or constructors
 - Useful for defining new structured types of data
 - Act similarly to primitive variables

```
struct tagname {  
    type1 name1;  
    ...  
    typeN nameN;  
};
```

```
// the following defines a new  
// structured datatype called  
// a "struct Point"  
struct Point {  
    float x, y;  
};  
  
// declare and initialize a  
// struct Point variable  
struct Point origin = {0.0, 0.0};
```

Using structs



- Use “.” to refer to a field in a struct
- Use “->” to refer to a field from a struct pointer
 - Dereferences pointer first, then accesses field

```
struct Point {  
    float x, y;  
};  
  
int main(int argc, char** argv) {  
    struct Point p1 = {0.0, 0.0}; // p1 is stack allocated  
    struct Point* p1_ptr = &p1;  
  
    p1.x = 1.0;  
    p1_ptr->y = 2.0; // equivalent to (*p1_ptr).y = 2.0;  
    return 0;  
}
```

simplestruct.c

Copy by Assignment



- You can assign the value of a struct from a struct of the same type – *this copies the entire contents!*

```
#include <stdio.h>

struct Point {
    float x, y;
};

int main(int argc, char** argv) {
    struct Point p1 = {0.0, 2.0};
    struct Point p2 = {4.0, 6.0};

    printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
    p2 = p1;
    printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
    return 0;
}
```

structassign.c

typedef



- Generic format:
- Allows you to define new data type *names/synonyms*
 - Both `type` and `name` are usable and refer to the same type
 - Be careful with pointers – `*` before `name` is part of `type`!

```
typedef type name;
```

```
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char *str;

// make "Point" a synonym for "struct point_st { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
    superlong x;
    superlong y;
} Point, *PointPtr; // similar syntax to "int n, *p;"

Point origin = {0, 0};
```

Dynamically-allocated Structs



- You can **malloc** and **free** structs, just like other data type
 - **sizeof** is particularly helpful here

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;    // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
    Complex* retval = (Complex*) malloc(sizeof(Complex));
    if (retval != NULL) {
        retval->real = real;
        retval->imag = imag;
    }
    return retval;
}
```

Structs as Arguments



- Structs are passed by value, like everything else in C
 - Entire struct is copied – where?
 - To manipulate a struct argument, pass a pointer instead

```
typedef struct point_st {  
    int x, y;  
} Point, *PointPtr;  
  
void DoubleXBroken(Point p)    { p.x *= 2; }  
  
void DoubleXWorks(PointPtr p) { p->x *= 2; }  
  
int main(int argc, char** argv) {  
    Point a = {1,1};  
    DoubleXBroken(a);  
    printf("( %d, %d) \n", a.x, a.y);    // prints: ( 1, 1 )  
    DoubleXWorks(&a);  
    printf("( %d, %d) \n", a.x, a.y);    // prints: ( 2, 1 )  
    return 0;  
}
```


Returning Structs



- Exact method of return depends on calling conventions
 - Often in `%rax` and `%rdx` for small structs
 - Often returned in memory for larger structs

complexstruct.c

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;    // imaginary component
} Complex, *ComplexPtr;

Complex MultiplyComplex(Complex x, Complex y) {
    Complex retval;

    retval.real = (x.real * y.real) - (x.imag * y.imag);
    retval.imag = (x.imag * y.real) - (x.real * y.imag);
    return retval; // returns a copy of retval
}
```

Pass Copy of Struct or Pointer?



- Value passed: passing a pointer is cheaper and takes less space unless struct is small
- Field access: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize
- For small structs (like `struct complex_st`), passing a copy of the struct can be faster and often preferred; for large structs use pointers

Q&A

